Υποστήριξη χρόνου εκτέλεσης για προσεγγιστικούς υπολογισμούς σε ετερογενή συστήματα./Runtime support for approximate computing on heterogeneous systems.

Spyrou Michalis

University of Thessaly

November 11, 2015

ACKNOWLEDGEMENTS

I would like to thank my advisors Nikolaos Bellas, Spyros Lalis and especially Christos D. Antonopoulos for their help and guidance throughout this work. It is due to their inspiration and continuous encouragement that I was able to successfully complete this work.

Also I would like to thank my family for their continuous support throughout these years.

This work has been supported by the ”Aristeia II” action (Project ”Centaurus”) of the operational program Education and Lifelong Learning and is co-funded by the European Social Fund and Greek national resources.

ABSTRACT

Energy efficiency is the most important aspect in nowadays systems, ranging from embedded devices to high performance computers. However, the end of Dennard scaling [2] limits expectations for energy efficiency improvements in future devices, despite manufacturing processors in lower geometries and lowering supply voltage. Many recent systems use a wide range of power managing techniques, such as DFS and DVFS, in order to balance the demanding needs for higher performance/throughput with the impact of aggressive power consumption and negative thermal effects. However these techniques have their limitations when it comes to CPU intensive workloads.

Heterogeneous systems appeared as a promising alternative to multicores and multiprocessors. They offer unprecedented performance and energy efficiency for certain classes of workloads, however at significantly increased development effort: programmers have to spend significant effort reasoning on code mapping and optimization, synchronization, and data transfers among different devices and address spaces.

One contributing factor to the energy footprint of current software is that all parts of the program are considered equally important for the quality of the final result, thus all are executed at full accuracy. Some application domains, such as big-data, video and image processing etc., are amenable to approximations, meaning that some portions of the application can be executed with less accuracy, without having a big impact on the output result.

In this MSc thesis we designed and implemented a runtime system, which serves as the back-end for the compilation and profiling infrastructure of a task-based meta-programming model on top of OpenCL. We give the opportunity to the programmer to provide approximate functions that require less energy and also give her the freedom to express the relative importance of different computations for the quality of the output, thus facilitating the dynamic exploration of energy / quality trade-offs in a disciplined way. Also we simplify the development of parallel algorithms on heterogeneous systems, relieving the programmer from tasks such as work scheduling and data manipulation across address spaces.

We evaluate our approach using a number of real-world applications, from domains such as finance, computer vision, iterative equation solvers and computer simulation. Our results indicate that significant energy savings can be achieved by combining the execution on heterogeneous systems with approximations, with graceful degradation of output quality. Also, hiding the underlying memory hierar-

chy from the programmer, performing data dependency analysis and scheduling work transparently, results in faster development without sacrificing the performance of the applications.

ΠΕΡΙΛΗΨΗ

Η ενεργειακή απόδοση είναι η πιο σημαντική πτυχή των σημερινών συστημάτων, που αποτελούνται από φορητές συσκευές μέχρι και υπολογιστές υψηλών επιδόσεων. Ωστόσο, το τέλος του Dennard scaling περιορίζει τις προσδοκίες για βελτίωση της ενεργειακής απόδοσης σε μελλοντικές συσκευές, παρά την κατασκευή επεξεργαστών σε χαμηλότερες γεωμετρίες και μειώνοντας την τάση τροφοδοσίας. Πολλά πρόσφατα συστήματα χρησιμοποιούν ένα ευρύ φάσμα της εξουσίας διαχείριση των τεχνικών, όπως DFS και DVFS, προκειμένου να εξισορροπηθούν οι απαιτητικές ανάγκες για υψηλότερη απόδοση με τις επιπτώσεις της επιθετικής κατανάλωσης ενέργειας και με τις αρνητικές θερμικές επιδράσεις. Ωστόσο, οι τεχνικές αυτές έχουν κάποιους περιορισμούς, όσον αφορά τα CPU intensive workloads.

Τα Ετερογενή Συστήματα εμφανίστηκαν ως μια πολλά υποσχόμενη εναλλακτική λύση έναντι των multicores και των πολυεπεξεργαστών. Προσφέρουν πρωτοφανείς επιδόσεις όσον αφορά την ενεργειακή απόδοση σε ορισμένες κατηγορίες φόρτου εργασίας, ωστόσο, με σημαντικά αυξημένη προσπάθεια ανάπτυξης: οι προγραμματιστές πρέπει να καταβάλλουν σημαντική συλλογιστική προσπάθεια, σχετικά με τη χαρτογράφηση και βελτιστοποίηση κώδικα, το συγχρονισμό και μεταφορά δεδομένων μεταξύ των διαφόρων συσκευών και χώρων διευθύνσεων.

Ένας σημαντικός παράγοντας που συμβάλλει στο ενεργειακό αποτύπωμα των τωρινών εφαρμογών είναι ότι όλα τα μέρη του προγράμματος θεωρούνται εξίσου σημαντικά για την ποιότητα του τελικού αποτελέσματος, έτσι όλα εκτελούνται με πλήρη ακρίβεια. Ορισμένοι τομείς εφαρμογών, όπως big data, βίντεο και επεξεργασία εικόνας, είναι δεκτικά σε προσεγγίστικούς υπολογισμούς, πράγμα που σημαίνει ότι κάποια τμήματα της εφαρμογής μπορεί να εκτελεστούν με λιγότερη ακρίβεια, χωρίς να υπάρχει μεγάλος αντίκτυπος στο τελικό αποτέλεσμα.

Σε αυτή την μεταπτυχιακή διπλωματική εργασία σχεδιάσαμε και υλοποιήσαμε ένα σύστημα χρόνου εκτέλεσης το οποίο λειτουργεί ως back end ενός μεταγλωττιστή και ενός προφιλερ, τα οποία συνθέτου ένα task-based meta-programming model χρησιμοποιώντας OpenCL. Δίνουμε την δυνατότητα στον προγραμματιστή να παρέχει προσέγγιστικές λειτουργίες που απαιτούν λιγότερη ενέργεια, αλλά και να δώσει στον προγραμματιστή την ελευθερία να εκφράσει τη σχετική σημασία των διαφόρων υπολογισμών για την ποιότητα του τελικού αποτελέσματος, διευκολύνοντας έτσι τη δυναμική αξιοποίηση της εξισορρόπησης ενέργειας / ποιότητας με ένα πειθαρχημένο τρόπο. Επίσης έχουμε απλοποιήσει την ανάπτυξη παράλληλων αλγορίθμων σε ετερογενή συστήματα, απαλλάσσοντας τον προγραμματιστή από την επιβάρυνση εργασιών, όπως ο χρονοπρογραμμα-

τισμός των εργασιών και ο χειρισμός των δεδομένων σε διαφορετικούς χώρους διευθύνσεων.

Αξιολογούμε την προσέγγισή μας, χρησιμοποιώντας μια σειρά από πραγματικές εφαρμογές, που προέρχονται από τομείς όπως ο οικονομικός, η όραση υπολογιστών, επαναληπτικές μαθηματικές μέθοδοι και προσομοιώσεις σε υπολογιστή. Τα αποτελέσματά μας δείχνουν ότι σημαντική εξοικονόμηση ενέργειας μπορεί να επιτευχθεί με το συνδυασμό της εκτέλεσης σε ετερογενή συστήματα και προσεγγιστικών αλγορίθμων, με όχι ιδιαίτερα εμφανή υποβάθμιση της ποιότητας του τελικού αποτελέσματος. Επίσης, κρύβοντας την ιεραρχία μνήμης από τον προγραμματιστή, εκτελώντας ανάλυση δεδομένων μεταξύ των tasks και χρονοπρογραμματίζοντας τις εργασίες με διαφάνεια, οδηγούμαστε σε ταχύτερη ανάπτυξη εφαρμογών χωρίς να θυσιάζεται η απόδοσή τους.

Every single one of us goes through life depending on
and bound by our individual knowledge and awareness.
And we call it reality. However, both knowledge and
awareness are equivocal. One's reality might be another's
illusion. We all live inside our own fantasies.

- Masashi Kishimoto

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1

INTRODUCTION

Moore's law [1] and Dennard scaling [2], resulted in exponential performance increases for the past three decades. However, present technologies can no longer sustain the previous advances in energy efficiency, leading the industry to move towards multicore processors and heterogeneous systems.

Heterogeneous systems appeared as a promising alternative to multicores and multiprocessors, and currently they dominate the Top500 and Green500 HPC lists. This shift in architecture created a new ecosystem both for applications and programmers, which offers great performance and energy improvements, but at the same time, it increased the development effort. Now programmers must spend a significant amount of time for optimization, synchronization and transfer data between different devices and address spaces. Still, however, building an exascale computer with current technology would lead to a machine that requires huge amounts of energy to function, making it highly inefficient.

One promising technique for minimizing energy consumption, is through approximate computing. Right now, all parts of the program are considered equally important to the output result's quality, thus all are executed at full accuracy. Previous work [3, 4, 5], however, has shown that there are several classes of applications which have parts that do not affect the output quality significantly. These parts can tolerate approximations, either by substituting a certain portion of the code with a less accurate one or even replace it with a default value.

This thesis focuses on a runtime system, which serves as the backend for the compilation and profiling infrastructure of a task-based meta-programming model on top of OpenCL. The main contributions of this MSc thesis are:

i) The design and implementation of an efficient, unified run-time support for heterogeneous architectures. It offers services for the computation execution, synchronization, memory management and power/performance monitoring of all components of the heterogeneous system.

ii) A memory manager which has two main goals: a) reduce the overhead introduced by memory management (allocations / deal-

locations) and b) ease data management and movement between the different address spaces of the heterogeneous system. The programmer does not have to create or free memory objects, nor has to explicitly transfer data between the different address spaces. Runtime creates each memory object, keeps track of it and reuses it, without the user's intervention and deallocates it only when it is necessary, for example when the device memory is full. Thus, we have a considerable effect on application performance and also we significantly ease the life of the programmer, as we remove a common burden when programming on heterogeneous systems, that of memory management.

iii) Concurrently exploit all available resources of a heterogeneous system. Present runtime systems, typically assign computationally heavy tasks to accelerators and the host remains idle whilst waiting for the results. Therefore, high utilization can oftentimes lead to bottlenecks on shared resources (memory hierarchy levels, disk etc.), so the runtime system continuously monitors the status of these resources and adapts computation scheduling and mapping in order to alleviate pressure on over-utilized shared resources.

iv) Automatic dependence analysis between different execution units of the application. Programming a heterogeneous system, requires the proper synchronization between execution units that use the same memory objects, or else this could lead to data corruption. This strain often leads the programmers to underutilize the heterogeneous system, as they usually synchronize every part their application in order to be safe. With dependence analysis, runtime executes concurrently every available part of the application, as long as there are available resources, thus maximizing the utilization of the system, and at the same time it postpones the execution of parts that are not ready.

v) Real-time monitoring of power and energy consumption for each accelerator of the heterogeneous system, in order to control the execution of computation and its mapping to the underlying heterogeneous architecture and reduce the power and energy footprint of the application within user-specified constraints. Also utilize the insight provided by compile-time analysis and profiling information, in order to aggressively reduce power and energy consumption and / or increase performance, even by executing computations at lower accuracy or totally skipping them if necessary.

The rest of the thesis is organized as follows. Chapter 2 presents required background about the Centaurus programming model and some technical background about power and energy measurements.

Chapter 3 presents the implementation of the runtime system and all the underlying functionality that is implemented. In chapter 4, we present the scheduling policies that are implemented. Chapter5, experimental evaluation is presented. Chapter 6 presents related work. Finally in chapter 7 I conclude my thesis.

2

BACKGROUND

This chapter introduces the key features of the task-based programming model that our runtime system supports and we explain each #pragma directive in detail, so the reader can establish a solid understanding about the runtime design decisions. Also we analyze the main functionality behind the Intel's RAPL [9] and Nvidia's NVML [8] interfaces, which are used to measure energy and power consumption at runtime.

2.1 CENTAURUS PROGRAMMING MODEL

Our programming model adopts a task-based paradigm using #pragma directives to annotate parallelism and approximations. Tasks are implemented as OpenCL kernels [6], containing both the accurate and the approximate (if available) version of the code. One of the main objectives of the programming model is to alleviate the programmer from common burdens when programming a heterogeneous system, such as inter-task synchronization, scheduling and data manipulation. Also, the programmer can explore the quality / energy tradeoff at runtime, by expressing wisdom on the importance of different parts of the code for the quality of the end-result.

Listing 2.1, summarizes the #pragma acl directives used for task manipulation. The task body specifies the accurate implementation of the task, defined as a function call, which corresponds to an OpenCL kernel. The approximate implementation of the task is provided by the programmer via the approxfun() clause. This is usually simpler and less accurate than the accurate version, however it has a lower energy footprint.

The significant() clause specifies the relative significance of the computation implemented by the task for the quality of the output, with a value (or an expression) in the range [0, 100]. If set to 100 or omitted, the runtime will always execute the task accurately. If set to zero, the runtime will always execute the task approximately, or even discard it if an approximation is not available.

The programmer must also specify the input and output parameters of each task, using the in(), out() and inout() clauses. This information is exploited by the runtime system in order to perform data flow analysis and data management as explained in Chapter3. Fur-

thermore, the programmer can force data transfers, that overwrite any existing data, from and to the device, using the device_out() and device_in() clauses respectively. Note, that both the approximate and accurate versions of the task, must have the same type and number of arguments as input and output. Finally, we also support array ranges, in the form of array[i:i+size] in the spirit of OpenACC [7], as arguments to in() or out() clauses to further reduce unnecessary data transfers.

Listing 2.1: #pragma acl task

```
#pragma acl taskgroup label( string_expr )
 [energy_joule( uint ) | ratio( double )]

#pragma acl task [approxfun( function )]
 [significant( expr )]
 [in( varlist )] [out( varlist )] [inout( varlist ) ]
 [device_in( varlist )] [device_out( varlist )]
 [device_inout( varlist )]
 [workers( int_expr_list )] [groups( int_expr_list )]
 [bind( device_type )] [label( "name" )]
 [bind_approximate(device_type)] [bind_accurate(device_type)]
 accurate_task_impl(...) ;

#pragma acl taskwait [label( "name" )]
```

Because the tasks are implemented as OpenCL kernels, the programmer must specify the number of work-items work-groups geometry for kernel execution. This is done via the workers() and groups() clauses, which follow the semantics of local and global work size of OpenCL, respectively. OpenCL kernels are able to run on every device on the heterogeneous system but sometimes the implementation is optimized for a specific device or would not be executed efficiently in certain devices. For those reasons, the programmer can explicitly associate a task for execution on a specific device using the bind() clause. A possible usage would look like this: bind(ACL_GPU), associating the task to a GPU. Also, if the user would like to provide an accurate or approximate version of a kernel, but wants this kernel only to be executed on a specific device (for example this kernel is optimized for GPU and executing it to CPU would result to an increased overhead), there are the special bind_accurate() and bind_approximate() clauses. This way, if the task is to be executed accurately, the runtime will not choose a device, rather respect the programmer's decision.

The programmer can associate a task with named task groups using the taskgroup directive and the label() clause. This action, associates a task with named task groups which are characterized by a unique string identifier. This way the programmer can classify tasks and provide information, such as ratio or an energy budget, that the

runtime system will use in order to make scheduling decisions. The ratio() clause accepts as an argument a value (or expression) ranging in [0.0, 1.0] which specifies the minimum percentage of tasks of the specific group that the runtime should execute accurately. Finally, the energy_ratio() clause takes as an argument a positive, non-zero value that sets the energy limit for this particular task group.

Lastly, the taskwait directive specifies an explicit synchronization point, acting as an execution and memory barrier. By default, taskwait waits on all issued tasks so far, unless the label() clause is present, which limits the explicit barrier only to tasks of the specific task group.

Listing B.1 depicts an example application, Molecular Dynamics (MD), that fully utilizes our programming model.

2.2 RUNNING POWER AVERAGE LIMIT AND NVML

Intel's RAPL(Running Average Power Limit) interface provides platform software with the ability to monitor, control, and get notifications on SOC power consumptions. In RAPL, platforms are divided into domains for fine grained control, as seen in figure 1. These domains include package, DRAM controller, CPU core (Power Plane 0), graphics uncore (power plane 1), etc.

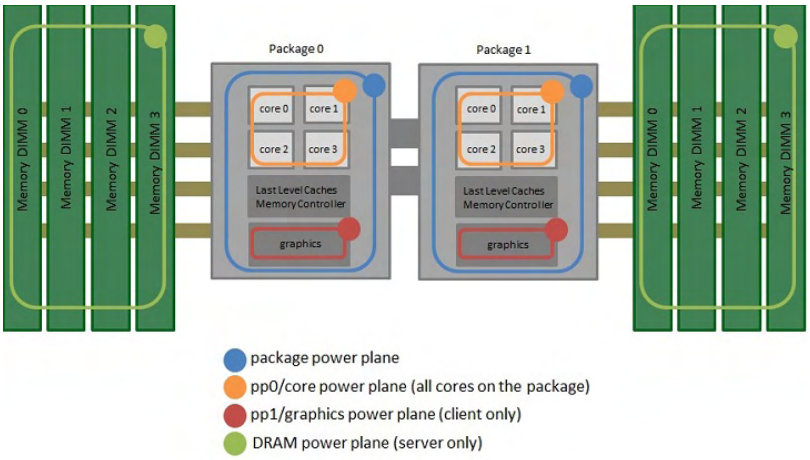

Figure 1.: RAPL power domains.

RAPL provides a set of counters providing energy and power consumption information and even give the user the option to set a power limit. RAPL is not an analog power meter, but rather uses a software power model. We can access these hardware counters through linux's msr interface.

Finally, recent NVIDIA GPUs can report power usage via the NVIDIA Management Library (NVML). Power reported is for the entire board

including GPU and memory and the reading is accurate to within a range of +/- 5 watts.

3

IMPLEMENTATION

This chapter dives into the implementation of the runtime system. First we analyze the general architecture and describe the lifetime of a task. Then we inspect the most important components of our runtime system: a) data management [3.2] , and b) data flow analysis [3.3]. Scheduling policies are described in chapter 4.

3.1 GENERAL ARCHITECTURE

Figure 2 outlines the general architecture of our runtime system. It is organized as a master/slave work-sharing scheduler. For each device on the system, two threads are created: (a) a memory transfers thread, that is responsible for transparent data transfers between the host and the device, and (b) a task issue thread, that is responsible for issuing tasks, which are implemented as OpenCL kernels, for execution to the corresponding device. Task has both the accurate and approximate OpenCL kernels pre-compiled and stored in a fat-binary, but we also support Just In Time (JIT) compilation if necessary. Our runtime reuses the underlying vendor OpenCL implementation for each device for data transfers, code execution, as well as to identify system configuration.

The master thread executes the main program sequentially and every task that is created, is first analyzed for possible dependencies between previous tasks, as explained in detail in section 3.3, and then gets stored either into the global "ready" pool either in the "not read" pool. The scheduler thread selects the last task that was inserted into the ready pool (LIFO), and then decides (section 4) on which device the task will be executed.

All task scheduling and data manipulation are transparent to the programmer, without having to manually transfer data to different address spaces. Thus, maximizing the utilization of the system is no longer an issue of the programmer, since our runtime can exploit every resource available as long as there is enough task parallelism. Also, it should be noted that the runtime tries to overlap data transfers with computations when possible by prefetching data for tasks to be executed while other tasks still keep the computational units of the device busy.

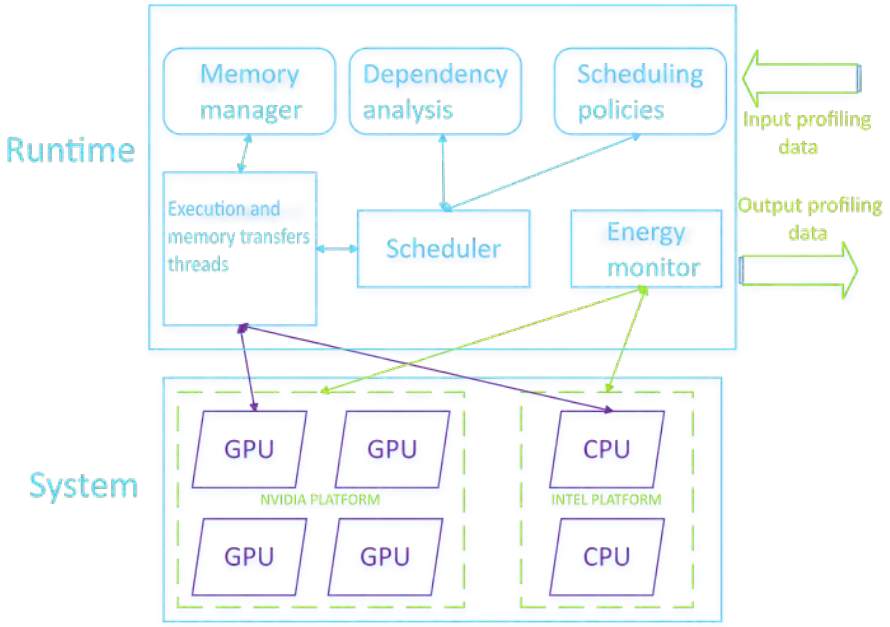

Figure 2.: Runtime architecture overview.

Our runtime can also extract all the information regarding task's execution, such as execution time and energy consumption and also data transfers time. This information can be digested by an offline tool, i.e. profiler, which will then formulate some functions that predict execution time and energy consumption. These functions can be fed back into the runtime in order to help it during different policies.

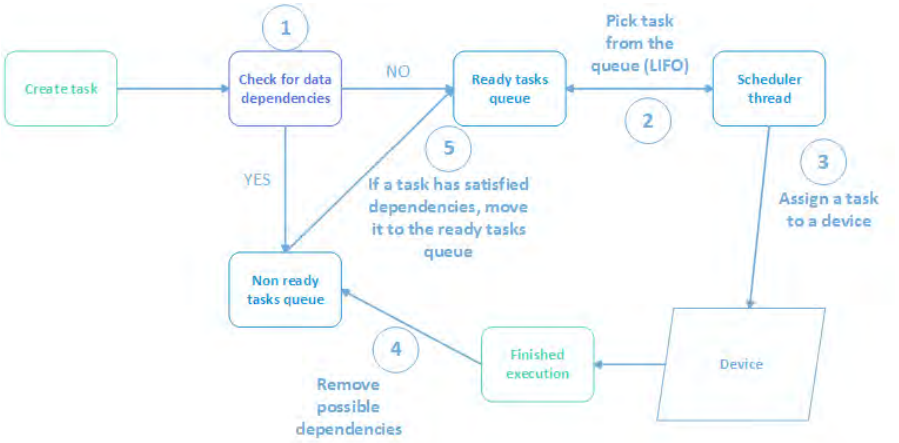

Figure 3.: Life of a task.

Figure 3, depicts the life of a task, from its creation to its execution and finally its destruction. The first action is to determine if the task

has dependencies with a previous issued task, that is still execution. If the task has no dependencies, we add it to the ready tasks pool from where the scheduler can select it for execution. If the task has one or more dependencies, we add it to the non ready tasks pool and we wait until these dependencies a resolved.

3.2 DATA MANAGEMENT

Due to the high number of different address spaces in a heterogeneous system, big emphasis was given to the creation of a data management unit, which is responsible for keeping track of every memory object that is created throughout the application's lifetime. Figure 4 depicts the design of the data management unit.

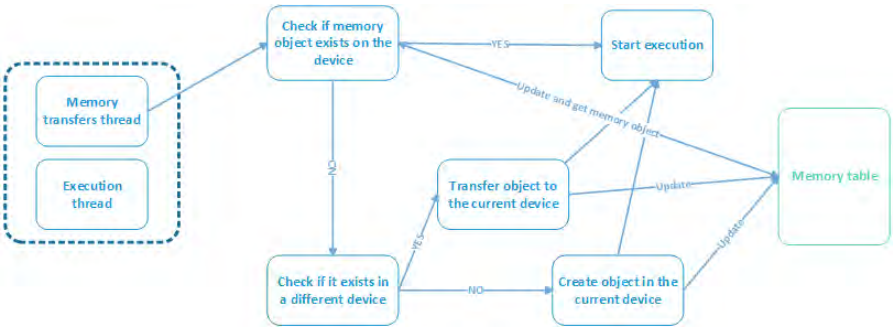

Figure 4.: Overview of data management unit.

When a device picks a task for execution, the first job of the memory transfers thread, is to transfer data to the corresponding device in order to start execution on the OpenCL kernel by the execution thread. Firstly, we check if a memory object of the current task, has already been created and thus it is stored in the memory table. If the answer is positive and the object resides on the current device, we reuse the memory object without having to allocate memory on the device or make redundant memory transfers.

In case where the memory object has been created but resides on a different device, we have to create a new one and copy the data from the either the remote device, or host. Note, that we cannot always copy the data directly from host, as the previous task that was executed may have altered them. For this reason, there are several states for the memory objects as seen in figure 5, so we can ascertain the action that we will take. The implementation is similar to a cache coherence protocol.

At first, every entry of the memory object table, stars from the **INIT** state. Then, when a new object is created and stored, the memory object goes to the **TRANSFERRING** state and stays there until the memory transfer is completed. Note that because we treat CPU

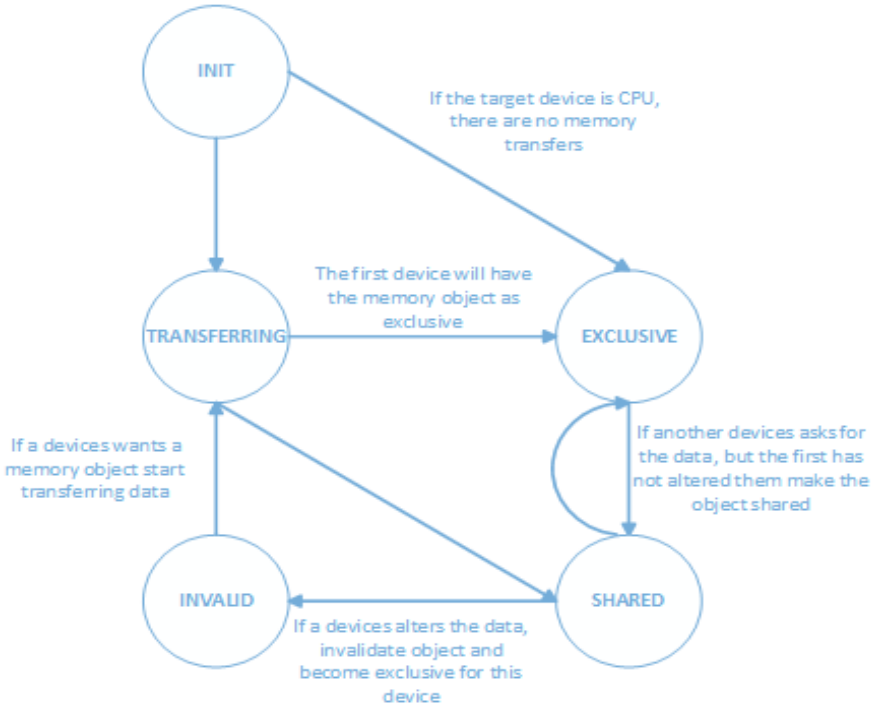

Figure 5.: Different states of a memory object.

as an accelerator, sometimes there isn't a TRANSFERRING state as there are actually no transfers between the host (CPU) and the device (CPU). Following up the termination of data transfer to the target device, the memory object enters the **EXCLUSIVE** state, meaning that it is used only by one device in the system. If an other device wants a copy of the same data, the memory object enters at first in the TRANSFERRING state and then becomes SHARED between the multiple devices.

The moment when a task alters the contents of a memory object, meaning that it used the out() clause in order to write the results on top of a previous created memory object, every other copy gets invalidated and it enters again the EXCLUSIVE state. Similarly, when a memory object is in the **INVALID** state and requires data from a different device it enters the SHARED state.

Runtime can also handle memory ranges in the form of array[i:size], minimizing this way the need for extra memory transfers between the host and the device. For each memory object, we keep track of the available data that exist on the device and each time we transfer, if necessary, only the sub-range of data that is needed in order to execute the task. Finally, it should be noted that runtime wraps every malloc(), realloc(), calloc() and free() call of the main program, in order to have full control of the host memory, keeping track of each

object's size and also freeing device memory when the host calls free() on the corresponding host buffer.

3.3 DATA FLOW ANALYSIS

One of the most important features of our runtime system, is the automatic data flow analysis at the granularity of tasks. Runtime exploits the information provided by the programmer via the in/out clauses and keeps track of the memory ranges read and written by each task. This knowledge is used for a) detecting data dependencies between tasks and b) automating memory transfers among different address spaces of the heterogeneous system.

It not uncommon for an application to have data dependencies between its tasks, as the data are usually reused in the scope of the application. For example, as shown in listing 3.1, task "calculate_vectors" writes its output in memory object "B1" and task "calculate_product" needs these data as input, in order to be executed. This of course is a simple scenario, but there are cases where the dependence graph can be very complicated, as seen in figure 6, and it proves to be a huge burden for the programmer to manage.

Listing 3.1: Example of dependencies between tasks.

```
#pragma acl task in(A1) out(B1)
calculate_vectors(A1, B1, size);

#pragma acl task in(B1) out(B2)
calculate_product(B1, B2, size);
```

For each new task that is created, runtime checks for potential WaW, RaW or WaR data dependencies between tasks that have not yet finished executing. Runtime keeps a dependency table, in order to identify these dependencies and enforce execution of inter-dependent tasks in the order they were spawned. Once the dependencies of the task are resolved, it is then transferred to the ready queue (3, step 5) and it can be selected for execution.

The dependency table is implemented as a hash table and an example is shown in table 1. The hash table data structure was selected mainly because of speed, as they generally have an average complexity of $O(1)$. The host address of each memory object is used as the hash key and as value there are two pools; the in and out pool, where tasks are stored. For any given task that is created, we search each memory object in the dependency table and we store the task to the corresponding pool. If a there are more than one tasks in the "out" pool, we know that there is a dependency, so the execution of the task is postponed. The task will be executed only when all dependencies are resolved, forcing this way the correct execution between tasks. Finally, when a task finishes execution, it removes itself from

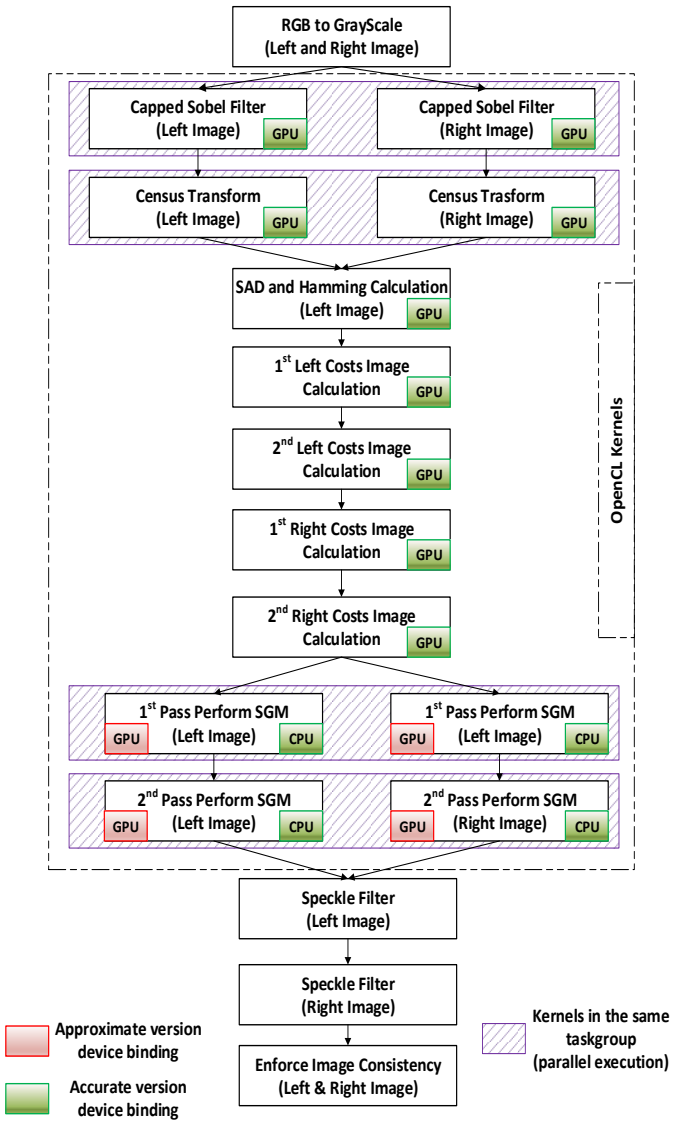

Figure 6.: SPStero Disparity's application graph.

the dependency table. Note, in the case of memory ranges, the dependency exists only if there is overlapping between the memory objects. Else, there is no dependency between the tasks, increasing this way the task parallelism.

key	value	
	in	out
&A1	calculate_vectors	NULL
&B1	calculate_product	calculate_vectors
&B2	NULL	calculate_product

Table 1.: Dependency table for the example 3.1.

4

SCHEDULING POLICIES

Every task includes the binaries of both accurate and approximate versions of the OpenCL kernels. When issuing the task for execution, the runtime decides whether it will execute the approximate or the accurate version. Also, the runtime must decide the proper device in order to execute the task. For example, enabling a new device for executing a new task, may result in high energy consumption which is not always desirable by the programmer. Thus, there is a number of different policies, each of them trying to achieve goals set by the programmer.

The programmer can either choose to minimize execution time, meaning that the runtime will use all the available resources of the system without caring about energy consumption, or he may want the application to have the lowest possible energy footprint. Furthermore, the programmer may also set an energy threshold on a specific task group, giving the runtime more freedom in order to choose the appropriate execution mode.

Achieving the objectives set by the programmer, without having prior knowledge of the application's behaviour, is a very difficult task. Fortunately, our runtime system can utilize information by prior profiling analysis 4.2, in order to predict execution time or energy consumption to some extent. Also, runtime keeps track of the average power for each device, so it can properly calculate the energy consumption of each task and data transfer. If profiling information are not available, runtime tries to predict the execution time and energy consumption based on the execution history of the tasks.

4.1 MEASURING ENERGY AND POWER CONSUMPTION

For each platform, we create a thread that monitors energy and power consumption using RAPL for Intel CPUs and NVML for Nvidia GPUs. Each thread periodically polls these interfaces and along with a timestamp, these values are stored in order to be accessed later by the profiler. At runtime, we keep the most recent power readings and when a task finishes execution, it calculates the energy it consumed.

Although, measuring the energy consumption for each task presupposes that we know the exact time each task started and finished

execution, something not trivial due to the asynchronous nature of our runtime. OpenCL events are able to provide us with this information, but only when kernels are executed on a single command queue. When multiple kernels are issued for execution on the same device, but in multiple command queues, OpenCL events do not provide the actual time the kernels started execution, rather provide the time they were issued on the device. Thus, in order to determine the actual kernel execution time for Nvidia GPUs, we use [4.1] the special register *clock* in order to measure the number of cycles that elapsed while the kernel was executing. For Intel CPUs we use the clock() function, in a similar way.

Listing 4.1: Function for reading clock register on Nvidia GPUs.

```
long long int clock_time()
{
    unsigned int clock_time = 0;
    long long int extended;
    asm("mov.u32 %0, %%clock;" : "=r"(clock_time) );
    asm("cvt.s64.s32 %0, %1;" : "=l"(extended) :
        "r"(clock_time));
    return extended;
}
```

4.2 PROFILING SUPPORT

Runtime can export detailed information about the execution of each application, if the profiling mode is enabled. Table 2 shows in detail the output that runtime exports, so an external application, i.e a profiler, could gather data about the application.

The profiler can then collect this information and perform a statistical analysis generating a prediction model for each kernel and taskgroup. This information can later be handed over to the runtime system in order to help it take decisions about the scheduling of the tasks and also choose the right execution mode (accurate/approximate). The output and input files are written and read as binaries and the specific input format can be seen in table 3. There are two prediction functions, one for predicting execution time and one for energy consumption, for each device on the system. These functions are mapped to a specific OpenCL kernel and to a specific taskgroup of the application.

4.3 MINIMIZING EXECUTION TIME / ENERGY CONSUMPTION

Given that data transfers typically incur significant overhead, data locality is one of the main criteria, along with resource availability, affecting scheduling of tasks to devices. Thus, when the scheduler

Field	Description
Name	Task's name and OpenCL's kernel name
Execution mode	Approximate/accurate
Start time	OpenCL timestamp of when the task was issued for execution
Stop time	OpenCL timestamp of when the task finished execution
Execution time	Task's execution time in ms
Input/Output number	Total number of input and output memory objects
Start time	OpenCL timestamp of start of memory transfer to/from the device
Stop time	OpenCL timestamp of end of memory transfer to/from the device
Transfer time	Memory object's transfer time in ms
Size	Memory object's size in bytes

Table 2.: Profiling information generated by the runtime.

Field	Description
Kernel Name	Execution time prediction function
	Energy consumption prediction function
Group Name	Execution time prediction function
	Energy consumption prediction function

Table 3.: Profiling information read by the runtime.

thread picks a new task for execution, the first action is to try to estimate the amount of time each data transfer would take for each device on the heterogeneous system. Then, with the help of the profiling information, we can make a solid prediction about the execution time both of the accurate and the approximate versions of the task. Also, in order to predict execution time, we have to take into consideration any other tasks that may be executing at that time on the device.

In both platforms,GPUs and CPUs, the most contributing factor in the execution time of two or more parallel tasks, is the geometry of the OpenCL kernel and specifically the number of blocks each kernel has. In figure (7), we can see that when a kernel spawns less blocks than the number of Streaming Multiprocessors (SMs), in this case 13, these kernels are run in parallel with small to no overhead, as long as they reside in a different OpenCL command queue. The same behaviour occurs in the case of one 8-core CPU, as seen in 8.

Thus, we can derive to equation (1a), that estimates the overlap time between two tasks and equation (1b), that estimates the execution time of the a $task_i$ based on profiling information. Function

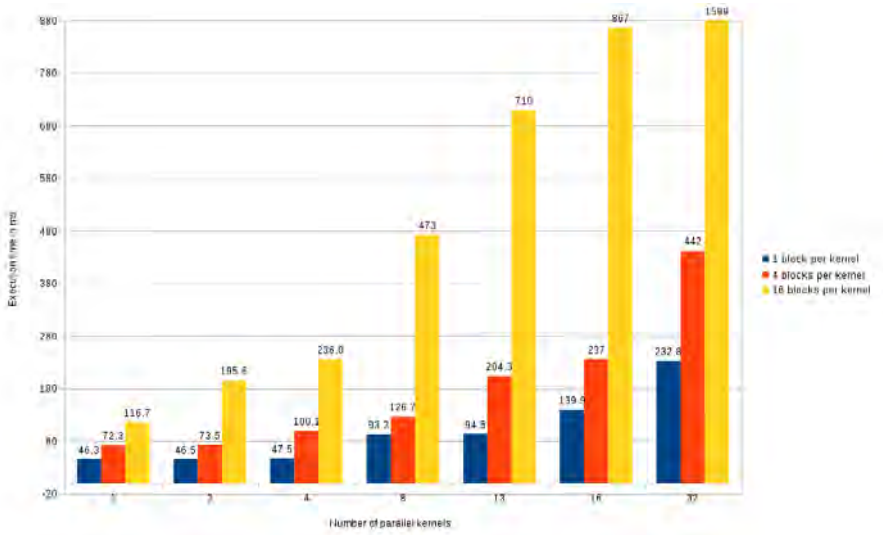

Figure 7.: GPU execution time for multiple kernels and 13 SMs.

pred_time(i) returns the estimated execution time of the task by the profiler, blocks(i) returns the number of (OpenCL) blocks this task has and SM is the number of Streaming Multiprocessors on a GPU or the number of cores on a CPU, that are available at that given moment.

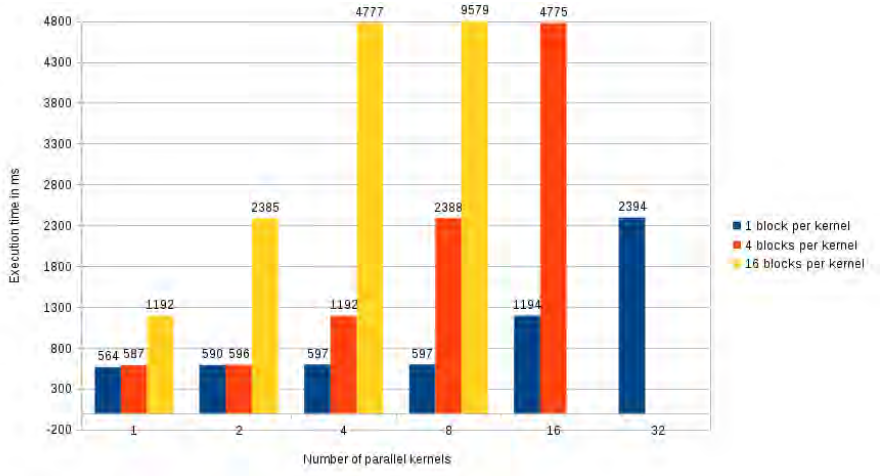

Figure 8.: CPU execution time for multiple kernels and 8 cores.

$$Overlap\_time(i) = pred\_time(i) * (\frac{blocks(i) \bmod SM}{blocks(i)}) \quad (1a)$$

$$Execution\_time = pred\_time(i) - Overlap\_time(i-1) \quad (1b)$$

After calculating the execution time for each device, we add the total data transfers time. Then we derive to the total time that the task will need in order to execute and transfer data on each device. Now, based on the policy that the user has selected, we can select the device that will execute the next task. If the user has selected to minimize the execution time, the device that has the lowest total time will be selected. In the case where the users has selected to minimize energy consumption, we have to compute the number of Joules the task will consume on each device, and select the one with the smallest value.

4.3.1 *Energy budget policy*

Our programming model gives the programmer the opportunity to set an energy budget for a specific task group. This way the execution mode is not determined based on the available energy we have left to spend. Listing 4.2 shows an example of a task group that has an energy budget of 300J and contains 4 tasks, each with different significance. The most important task of the algorithm 1, is to keep the output quality as high as possible, given the target energy budget. At first, the runtime tries to minimize the execution time of the group, by distributing the tasks across the system's devices as seen in line 12. If the energy target is not met, runtime moves each task, starting with the less significant one, to the device with the smallest energy consumption (line 22). If this step is not successful for meeting the desired target, runtime starts to approximate tasks, again starting from the least significant ones (line 25). Finally, if the energy budget is set too low, runtime starts dropping tasks until the target is met (line 27). The energy budget is valid until the next task wait. Then the group's energy budget is reset, except if the programmer sets a new value.

Listing 4.2: Setting energy budget on a task group

```
#pragma acl taskgroup label("group1") energy_joule(300)

#pragma acl task label("group1") in(A1) out(B1) significant(75)
calculate_vectors(A1, B1, size);

#pragma acl task label("group1") in(A2) out(B2) significant(25)
calculate_product(A2, B2, size);

#pragma acl task label("group1") in(A3) out(B3) significant(50)
calculate_vectors(A3, B3, size);

#pragma acl task label("group1") in(A4) out(B4) significant(50)
calculate_product(A4, B4, size);

#pragma acl taskwait
```

Algorithm 1 Energy budget algorithm

```
1: function CHECK_TARGET(energy_budget, current_energy)
2:     if energy_budget ≥ current_energy then
3:         exit
4:     end if
5: end function
6:
7: procedure ENERGY_BUDGET_POLICY
8:     NewGroup ← Sort all tasks in Group in ascending order of
   significance
9:     for each task_i in Group do
10:        for each device_i do
11:            Energy(i,j) = estimate_energy(i,j)
12:            device_i = shortest_time(i)
13:            current_energy = current_energy + Energy(i,device_i)
14:        end for
15:    end for
16:
17:    State = Change device
18:    while check_target(energy_budget, current_energy) do
19:        for each task_i in NewGroup do
20:            switch State do
21:                case Change device
22:                    device_i = min(Energy(i))
23:                    Next State = Approximate task
24:                case Approximate tasks
25:                    approximate(task_i)
26:                    Next State = Drop tasks
27:                case Drop tasks
28:                    drop(task_i)
29:            update_current_energy()
30:            check_target(energy_budget, current_energy)
31:        end for
32:        State = Next State
33:    end while
34: end procedure
```

5

EXPERIMENTAL EVALUATION

In this chapter we evaluate the runtime system with real life applications, such as PBPI, SPStereo Disparity etc., and with special benchmarks that are written in order to stress and check the functionality of the runtime system. First, we check the overhead, both in time and energy consumption, that our runtime introduces when executing an OpenCL application. Then we evaluate the runtime policies that exist in order to meet the user requirements.

The experimental evaluation was carried out on a system equipped with two Intel XEON E5 2695 processors, clocked at 2.3 GHz, with 128 GB DRAM and two Nvidia Tesla K80 GPUs. The operating system is Ubuntu 14.04, using the 3.16 Linux kernel. The GPU power monitoring interface returns the instantaneous power consumption polled every 2 ms.

5.1 UNIT TESTING BENCHMARKS

Before evaluating our runtime with real life applications, we developed a number micro benchmarks that test the runtime's functionality. Our first and most complex test, creates a number of tasks that has the dependency graph shown in figure 9. Each task does a simple addition and forwards the result to the next, until the result is returned to the host via "task F". In order to test the dependency analysis, there is no taskwait between tasks, only after the last one. We expect our runtime to discover all dependencies and force the correct execution between tasks.

Also, tasks are executed across different devices, testing at the same time the runtime's memory manager. Memory manager must move data across the different address spaces correctly, or else the results will be inaccurate. For example, task B1 executes on GPU2 and task B2 executes on GPU3. When task C will need the input data, in order to be executed at the CPU, memory manager transfers data from GPU2 and GPU3 to the CPU. Various other micro-benchmarks were created that test WaW, RaW and WaR dependencies and also test the correct data movement across different address spaces.

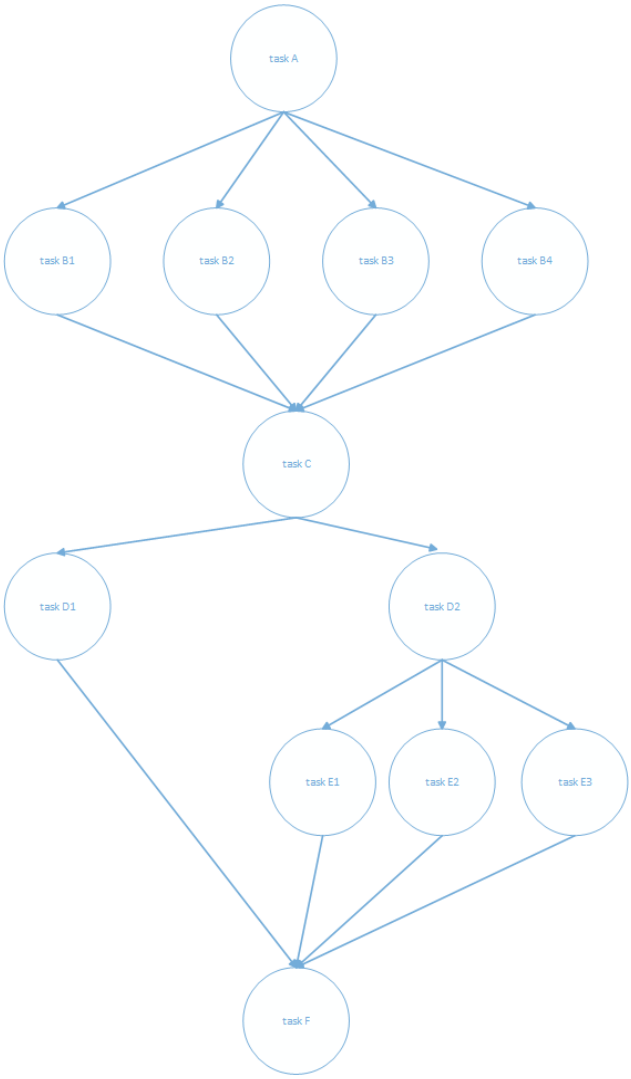

Figure 9.: Dependency graph of our unit test.

5.2 RUNTIME OVERHEAD

In order to measure the overhead that our runtime system introduces, compared to pure OpenCL implementations, we measured the execution time of some real life applications in their pure OpenCL form and then the execution time on our runtime system. All applications were executed in one device only, either CPU or GPU, without utilizing multiple devices. The same was done for the energy consumption. The results are presented in Figure 10 and a short description of each application can be fount in Appendix A. Table 5.2 presents the raw numbers for more clarity.

As we can see, the overhead introduced by our runtime system is negligible in most applications and some are even faster than the original OpenCL implementation. This happens for two reasons: a) our runtime system utilizes multiple command queues without the user's

intervention, taking advantage of possible task parallelism even on the same device, such as in the case of Molecular Dynamics (MD) and b) our runtime system automatically exploits all opportunities for minimization of data transfers and their overlap with computations. Note that SPStereo Disparity's implementation uses both CPU and GPU by default. The energy consumption overhead for each application follows the same pattern as with the execution times.

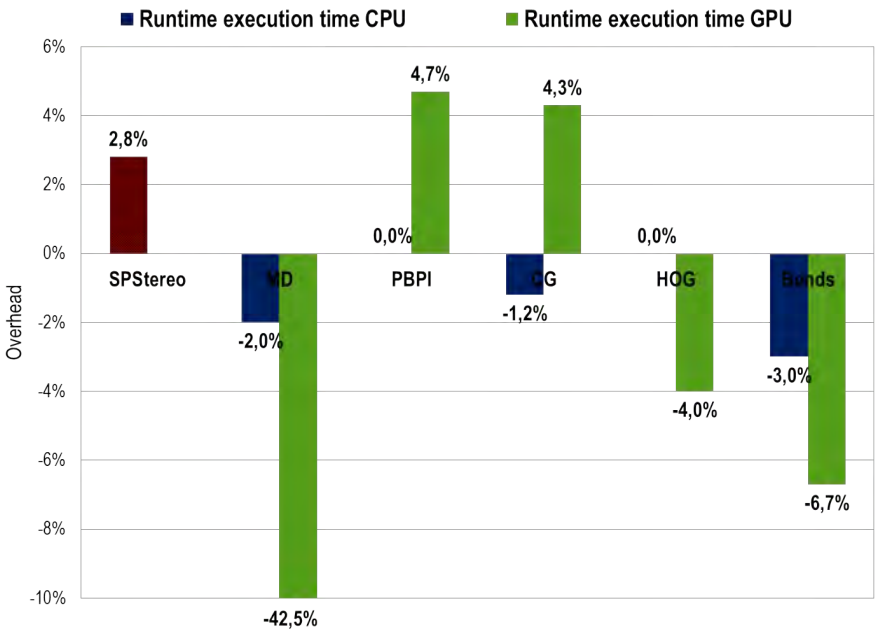

Figure 10.: Performance overhead w.r.t. the corresponding OpenCL implementation.

	SPStereo	MD	PBPI	CG	HOG	Bonds
Pure OpenCL (CPU)	4.98	309.2	53	85	15.13	2.05
Pure OpenCL (GPU)		146	85	146	3.95	1.91
Runtime (CPU)	5.12	294.51	53	84	15.13	2.04
Runtime (GPU)		84	89	46	3.8	1.79

Table 4.: Execution times between the pure OpenCl implementation and the runtime system. Time is in seconds.

5.3 SCHEDULING POLICIES

In order to evaluate our scheduling policies, firstly for the device distribution, we used applications with enough task parallelism and with only one execution mode (accurate), so our runtime could express wisdom on the decisions regarding the selection of a device. These applications were: MD B.1, PBPI and a matrix multiplication application written as a benchmark to test the runtime system. MD and PBPI have four tasks with no dependencies between them, and matrix multiplication spawns 100 independent tasks that each calculates a 2048*2048 float matrix multiplication. Table 5.3 presents the execution time and the energy consumption of each application for a different number of device configurations, each distributing the tasks across the corresponding number of devices. We can observe that for all applications, times on CPU are much greater than the time on GPU, and this has a negative effect on the total energy consumption. We expect our policies to find the best configuration in each case from all the possible ones.

Configuration	Matrix mul		MD		PBPI	
	Time	Energy	Time	Energy	Time	Energy
CPU	72 s	1823 J	156 s	27612 J	154 s	33132 J
1 GPU	18.01 s	2378 J	59.37 s	8011 J	50.6 s	5364 J
2 GPUs	10.95 s	2737 J	38 s	9880 J	42.2 s	8033 J
3 GPUs	7.99 s	3116 J	30 s	9960 J	36.9 s	9616 J
4 GPUs	6.3 s	3276 J	25.2 s	11188 J	28.5 s	9713 J

Table 5.: Execution time in seconds and energy consumption in Joules for different device configurations.

Figures 11 and 12 show the distribution of tasks for each application, when we executed the with the two policies. Each application has been profiled and profiling data are given to the runtime for predicting execution time and energy consumption for each task and task group. For MD and PBPI, runtime selects the optimal configuration for both cases. For the benchmark, runtime selected to execute some tasks to the CPU even if the execution will be much slower for these individual tasks, but overall due to the high number of task parallelism, this delay is hidden and the overall application runs slightly faster.

Runtime can also take decisions without profiling information, by keeping history for each task and device. First, runtime sends tasks to each device in order to measure execution time and energy, and then with these data starts to form a model that tries to estimate execution time and energy consumption. For applications like MD and PBPI, where all tasks have similar execution times, runtime does a good job distributing the tasks accordingly.

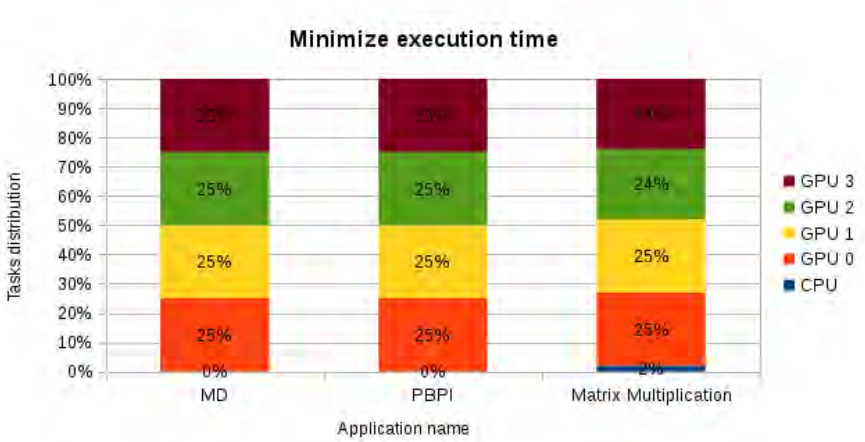

Figure 11.: Applications' task distribution for minimizing execution time policy.

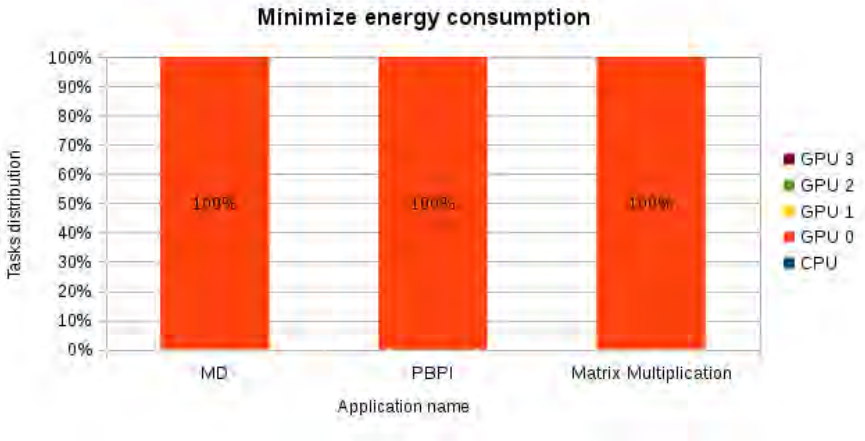

Figure 12.: Applications' task distribution for minimizing energy policy.

Figures 13 and 14, show the task distribution in each device, without profiling information. For the first policy, minimize execution time, runtime performs idealy, as the tasks have similar execution times. For the second policy, minimize energy consumption, runtime is close to the ideal execution, executing the 74% and 60% of the MD and PBPI tasks respectively, to one GPU. Power measurements are not stable at the beginning of the applications execution, thus, a significant amount of time is spend testing different configurations. For the matrix multiplication benchmark, runtime does not have any power measurements as the application spawns 100 tasks instantly.

Next, in order to evaluate the policies further, we tested all the applications again, but this time the runtime was free to decide the

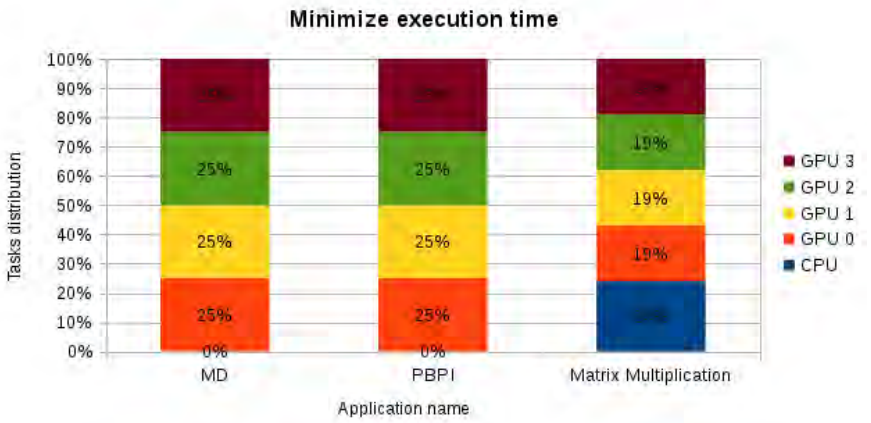

Figure 13.: Applications' task distribution for minimizing execution time policy, without profiling information.

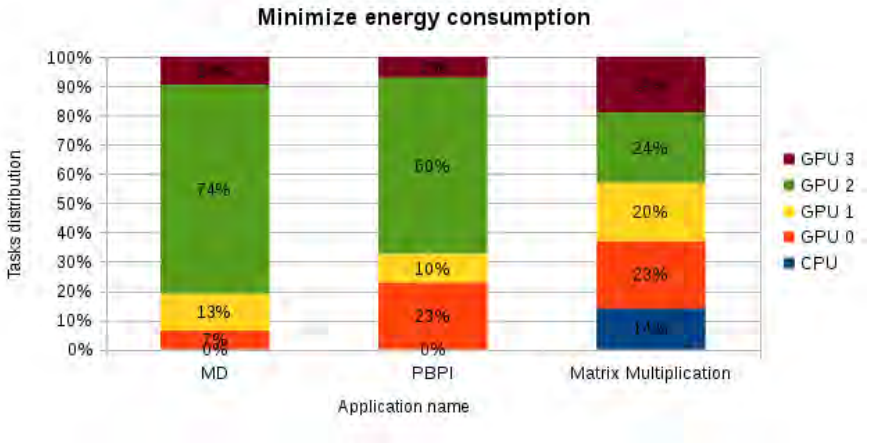

Figure 14.: Applications' task distribution for minimizing energy policy, without profiling information.

execution mode (accurate/approximate). Figures 15 and 16 show the execution time and output quality for each application, for the first policy. As the accurate execution time and energy, we used the results we got from the same policy using the accurate version of the tasks. We can observe that selecting the approximate kernel for each tasks, results in a huge drop in execution time, without affecting the output quality significantly.

For our second policy, we used again as reference the fully accurate version of the tasks. The results are shown in figures 17 and 17.The same behaviour is seen as with the first mainly because the applications does not have enough task paralelism. In the cases though, were there are multiple tasks, we can observe energy gains of even

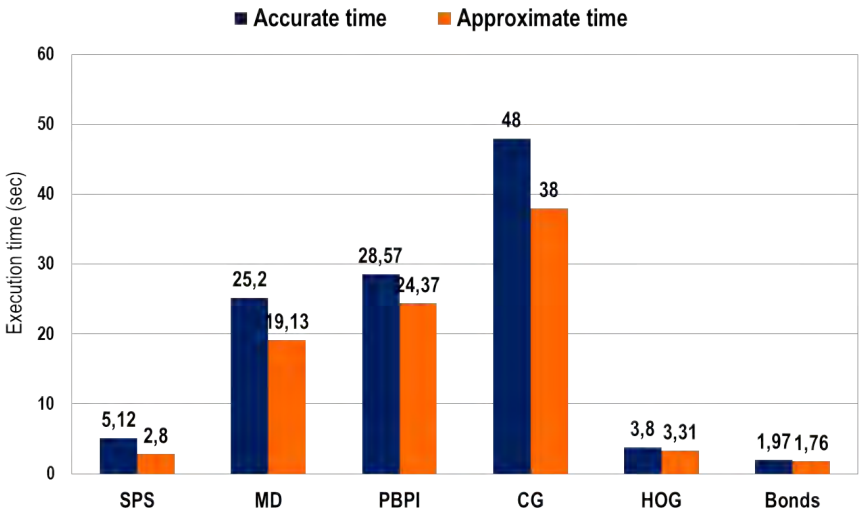

Figure 15.: Execution time difference for each application, using the minimize execution time policy.

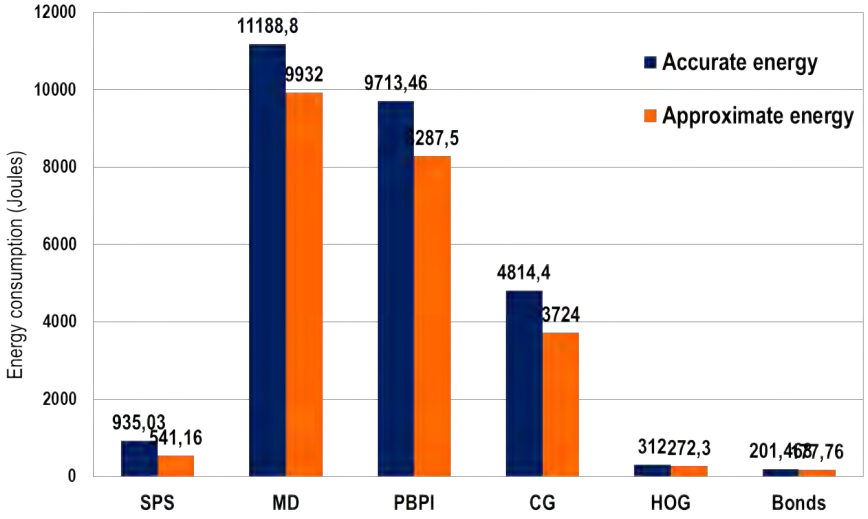

Figure 16.: Energy difference for each application, using the minimize execution time policy.

52% (MD) between the accurate and the approximate execution. Only for the applications MD and PBPI we can observe a difference in execution time between and energy between the two policies. The increased time for the minimize energy consumption policy, is due to the fact that all tasks are executed on one device (one GPU), but we can see that this results to a very low energy consumption.

Finally, we compare the execution time and energy consumption of MD, MM and PBPI for each policy for the case that we have profiling

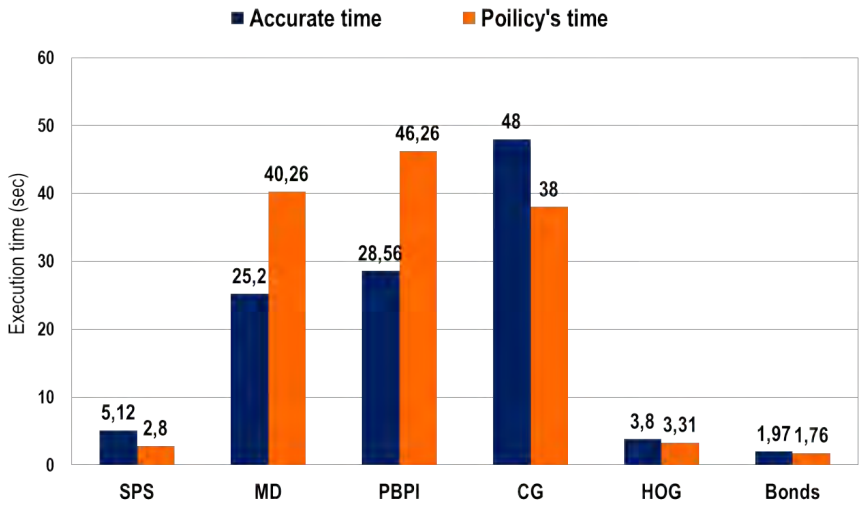

Figure 17.: Execution time difference for each application, using the minimize energy consumption policy.

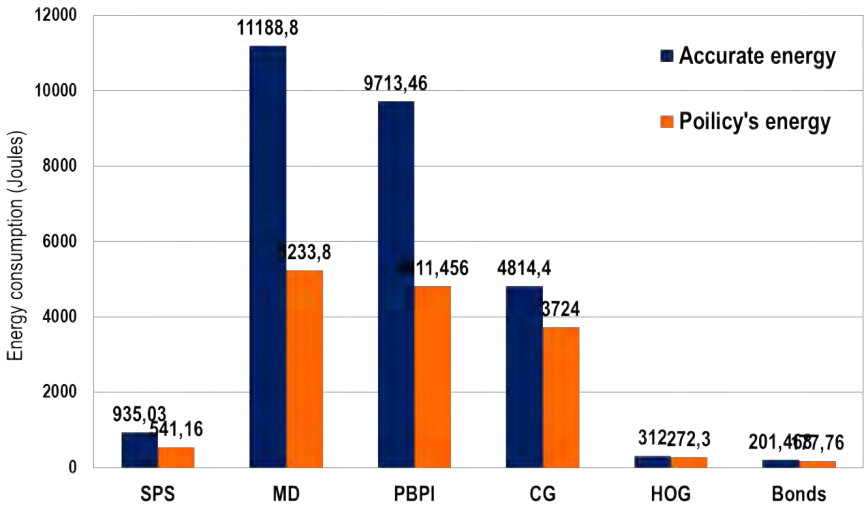

Figure 18.: Energy difference for each application, using the minimize energy consumption policy.

data and for the case we don't. Results can be seen in figures 19 and 20. We can observe that having offline profiling information is critical, if we want to achieve optimal performance, or else there is a significant overhead until the runtime learns the application's behaviour.

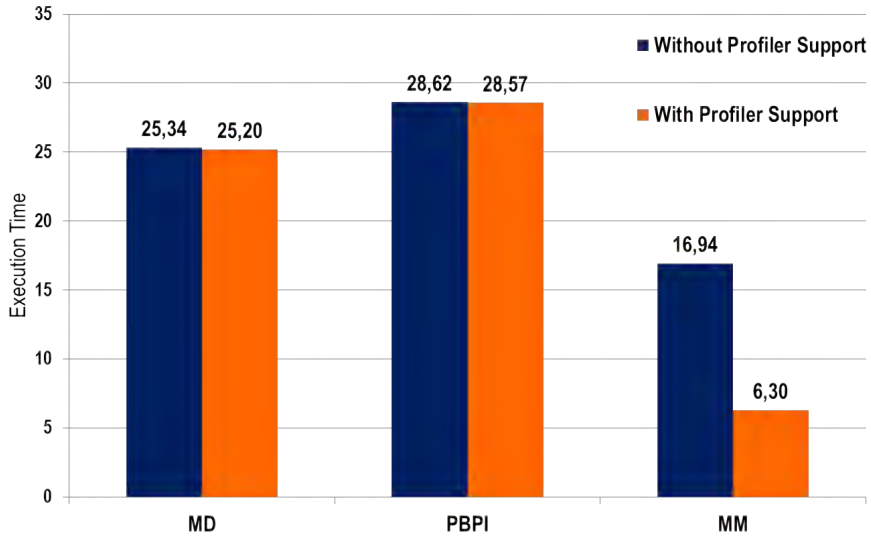

Figure 19.: Execution time with and without offline profiling information for each application, using the minimize energy consumption policy.

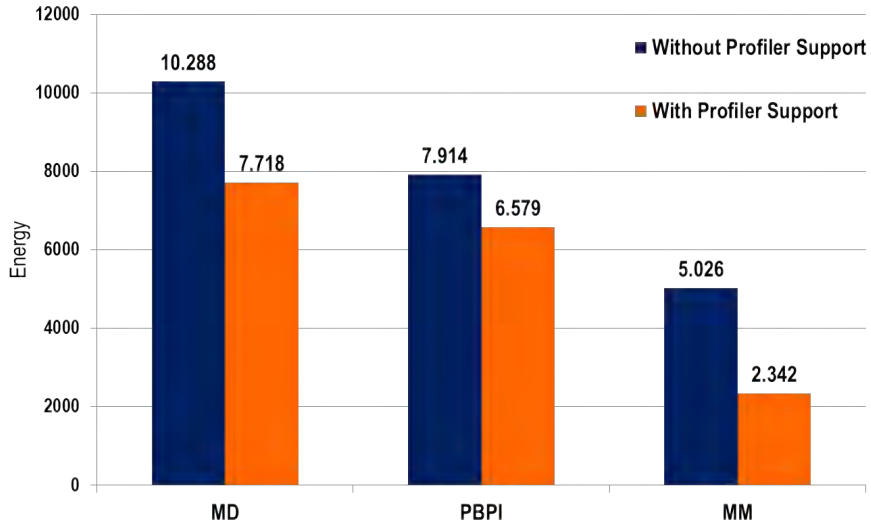

Figure 20.: Energy consumption with and without offline profiling information for each application, using the minimize energy consumption policy.

5.4 POLICY BASED ON ENERGY BUDGET

For our energy budget policy, we compute the energy that each group needs in order to be executed accurately, and then we measured the execution time and the output quality of the whole application by using a different energy budgets for each task group. Firstly we want

to test how runtime responds to random changes in energy budget. Thus, we execute an application (MD) and in each taskgroup we set a randomly distributed value between 1 and 13. Figure 21 shows the runtime actual execution energy for each taskgroup vs the target energy budget. We can observe that for energy budgets below 2 Joules runtime cannot make a valid decision and drops all tasks, leading to zero energy consumption.

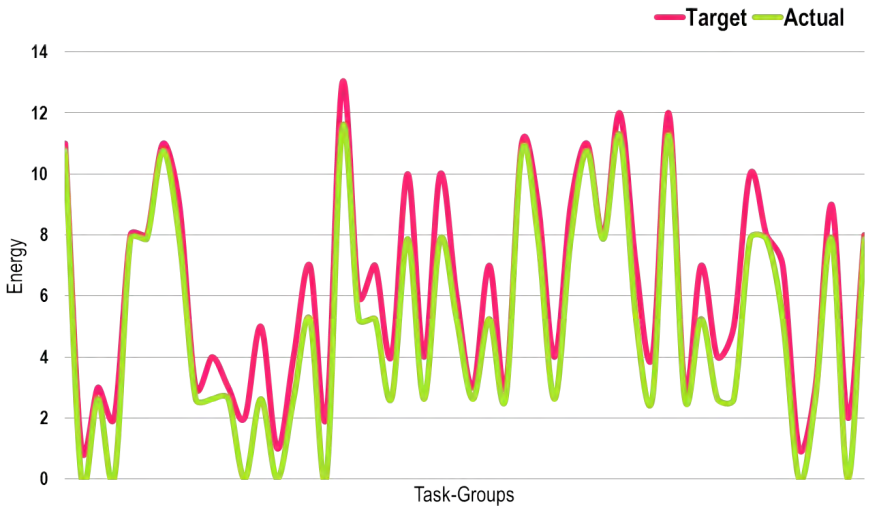

Figure 21.: Runtime's response to different energy budgets, with the energy budget algorith.

As some may observe, runtime is always below limit and sometimes even 30% off. Thus for better results, the energy that we save from each iteration, is passed down to the next. This way may surpass the energy budget for a particular taskgroup but we don't go past the total energy budget for the entire application. Runtime adaption for this algorithm is shown in figure 22.

Next, we tested our policy with a real life application and compare the output quality for each energy budget. Figure 23 show the output quality and execution time for the application MD, given that each group needs 12.41 Joules in order to be executed accurately. Quality below 96% is not accepted.

Subsequently, for each iteration, instead of keeping the energy budget constant, we modify it. This way some task groups will be executed accurately, some approximately and some will be dropped. We tried a different combination between accurate, approximate and dropped tasks, and the results are shown in figure 24. We can observe that MD is very tolerant in approximations, mainly because it simulates a system in equilibrium state and the total energy is preserved, even if some tasks are not executed.

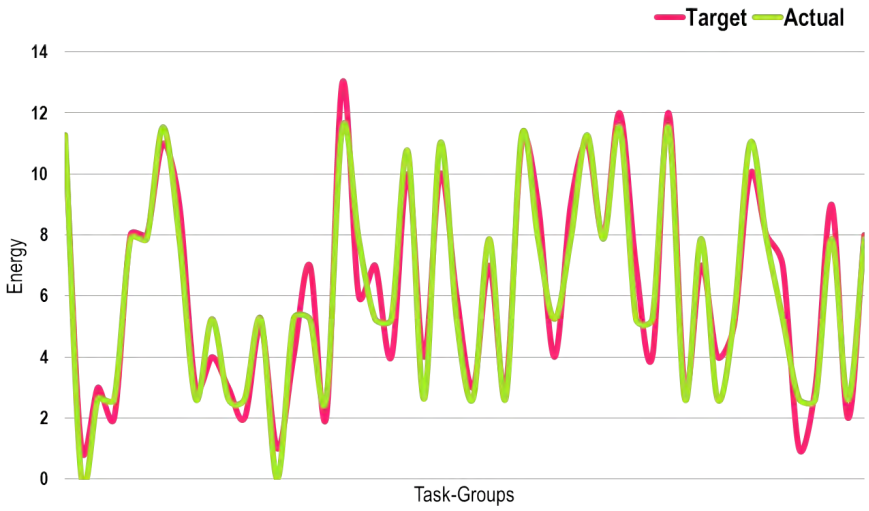

Figure 22.: Runtime's response to different energy budgets when saving the excess energy from previous iterations.

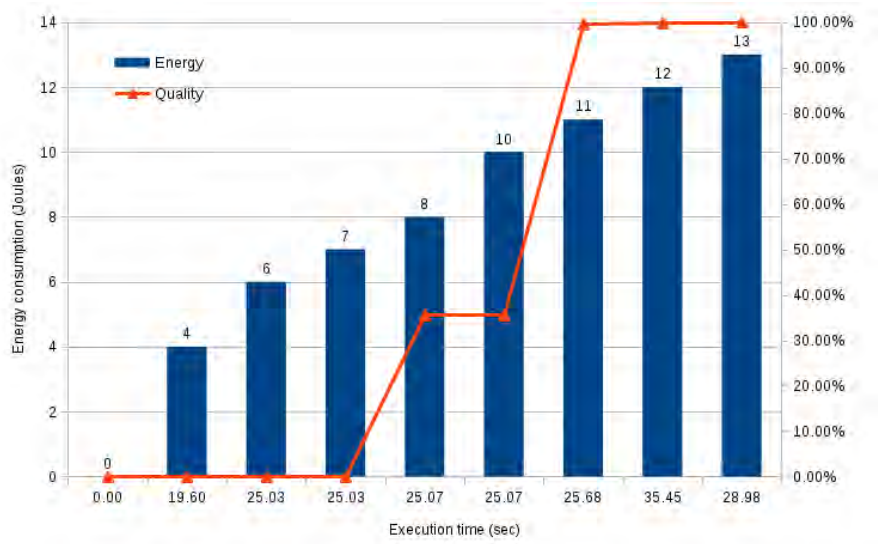

Figure 23.: Execution time, energy and quality of results for the application Molecular Dynamics, for different energy budgets.

PBPI on the other hand, is only tolerant to approximate execution of tasks. If we try to drop some kernels, the quality of result is dropped to 0. Also, if every task in PBPI's taskgroup is to be executed accurately, we need 0.32 Joules, a value so small that the user cannot set a valid energy budget.

Finally, the user can also set a ratio along with an energy budget. By doing so, the algorithm tries to meet the energy budget but can

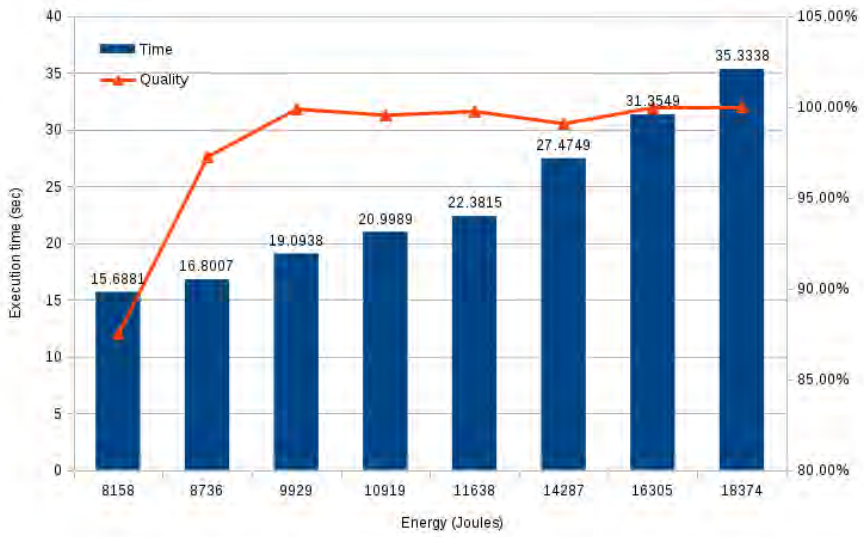

Figure 24.: Execution time, energy and quality of results for the application Molecular Dynamics, using different combination of energy budgets within an full application run.

only change the execution mode of the tasks that are below the the target ratio. More formally, if the taskgroup has n tasks and ratio is R, the $R * n$ least significant tasks are able to change it's execution mode (accurate, approximate, drop) in order to meet the energy budget. Of course, this means that runtime cannot always be below budget, if for example ratio is too high or energy budget is too low. Figures 25 and 26 show the runtime's response (red line) to a cumulative energy budget of 6737 Joules, for ratios 0.5 and 0.25 respectively. We can see that our runtime cannot meet the energy budget set by the programmer if she provides a relatively small amount of energy and high ratio. On the other hand for a cumulative energy budget of 9560 Joules, we can observe in figures 27 and 28 that the energy budget policy stays close to the real (blue) budgets set by the programmer.

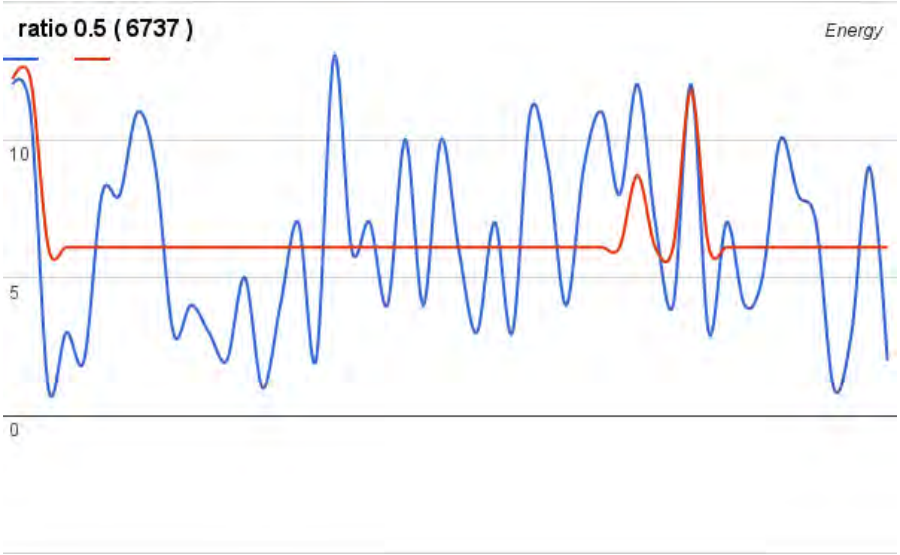

Figure 25.: Runtime's responce for ratio 0.5 and cumulative energy budget of 6737 Joules.

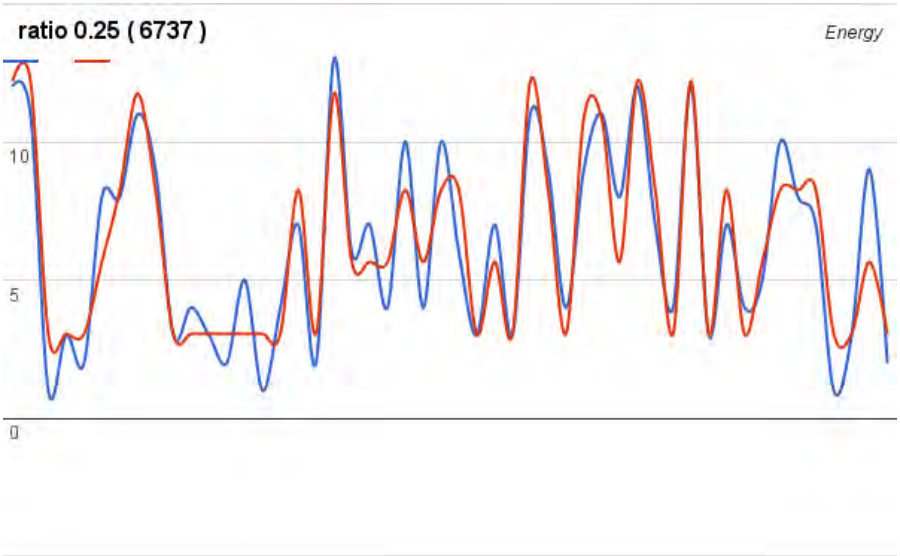

Figure 26.: Runtime's responce for ratio 0.25 and cumulative energy budget of 6737 Joules.

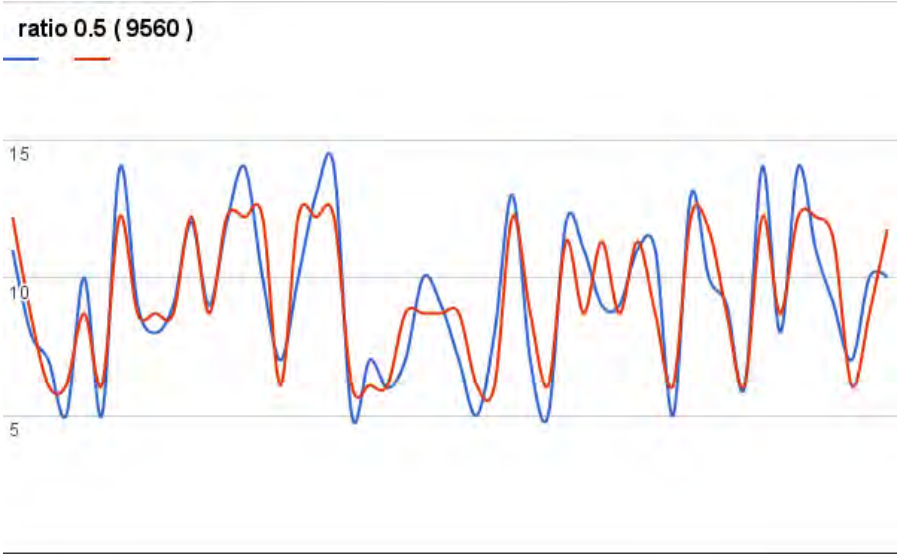

Figure 27.: Runtime's responce for ratio 0.5 and cumulative energy budget of 9560 Joules.

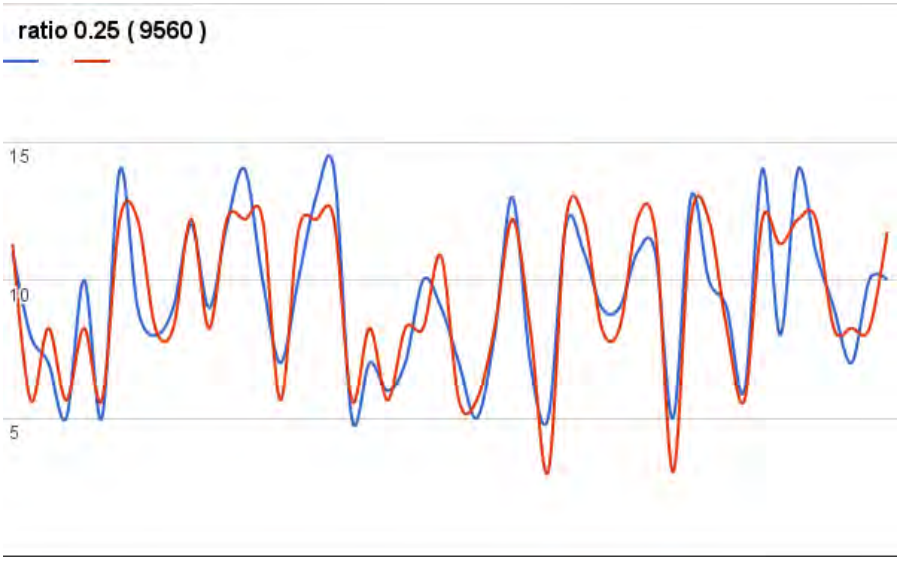

Figure 28.: Runtime's responce for ratio 0.25 and cumulative energy budget of 9560 Joules.

6

RELATED WORK

Green [3] is a framework for supporting energy-conscious programming using controlled approximation while providing QoS guaranteed to the programmer. Ringenburg et al. [16] proposes an architecture and tools for autotuning applications that enable trading quality of results and energy efficiency, assuming approximations on hardware level. EnerJ [15] proposes an approximate type system using code annotations without defining a specific programming and execution model. ApproxIt framework [17] approximates iterative methods at the granularity of one solver iteration.

Variability-aware OpenMP [18] also follows a #pragma-based notation and correlates parallelism with approximate computing. Quickstep [19] is a tool that parallelizes sequential code. It approximates the semantics of the sequential code by altering data and control dependencies. SAGE [20] is a domains pecific environment with a compiler and a runtime component that automatically generates approximate kernels in image processing and machine learning applications. GreenGPU framework [21] dynamically splits and distributes workloads on a CPU-GPU heterogeneous system, aiming to keep busy both sides all the time, thus minimizing idle energy consumption. It also applies DFS for the GPU core and memory for maximizing energy savings. Tsoi and Luk [22] estimate performance and power efficiency tradeoffs to identify optimal workload distribution on a heterogeneous system.

Our work introduces the concept of computation significance as a means to express programmer wisdom and facilitate the controlled, graceful quality degradation of results in the interest of energy eficiency. We support approximate computing in a unified, straightforward way on different devices of accelerator-based systems, thus exploiting and combining energy eficiency benefits from both heterogeneity and approximation. Our approach does not require hardware support apart from what is already available on commodity processors and accelerators.

7

CONCLUSION

In this MSc thesis we introduce a runtime system for supporting approximate computing on heterogeneous systems. We allow the user to express her wisdom on the importance of different computations for the quality of the end result, to provide approximate, more energy efficient implementations of computations and to control the quality / energy efficiency trade-off at execution time, using a single, simple knob. In addition the programmer can simply express her desire for minimizing the energy consumption of the entire application or minimize the execution time, leaving the rest to the runtime.

Also, our runtime eliminates the some technical concerns when programming a heterogeneous system, such as computation scheduling and data management, which are often a huge programming burden that limit productivity. We evaluated our implementation, not only with benchmarks that were designed to stress specific parts of the runtime system but also with real life applications and found that exploiting the concept of significance at the application level enables measurable energy gains through approximations, while the programmer maintains control of the quality of the output.

A key direction for future work, is to expand our runtime system, in order to support FPGAs and embedded systems, increasing this way the available platforms and also surge the possible options for approximations and minimizing energy.

Appendices

A

APPLICATIONS DESCRIPTION

A.1 PBPI

PBPI [11] is a high performance implementation of Bayesian phylogenetic inference method for DNA sequence data. It starts from random phylogenetic trees and estimates the likelihood of them being realistic. The trees are then modified and re-evaluated in an iterative evolutionary process. The tree with the maximum likelihood is the output of the application. The process is quite sensitive to errors and applying approximations is not a straightforward task.

A.2 HOG

HOG [10] is a computer vision application for pedestrian recognition using machine learning techniques. The algorithm divides the picture into independent blocks that can be computed in parallel. For each block a set of kernels is applied in pipeline manner. Initially, the first kernel creates a histogram of the gradients orientation. Then it combines them into a descriptor and finally feeds it on a Support Vector Machine (SVM) which classifies each block.

A.3 CG

The conjugate gradient (CG) is an iterative algorithm that is applied to systems of linear equations in order to acquire the numerical solution. The matrix of these systems is symmetric and positive-definite. The algorithm stops when reaches convergence within a tolerance value. We approximate some double floating point computations using the mixed precision technique, while keeping the last iterations accurate in order to reduce the relative error of the final solution.

A.4 SPSTEREO DISPARITY

SPStereo Disparity [12] is an application which calculates a dense depth estimate image from an input of a stereo pair camera. It consists of two parts. First it produces an initial disparity image. After-

wards, using the disparity image it exploits shape regularization in the form of boundary length while preserving connectedness of image segments to produce the depth estimate image. The most time-consuming part of the algorithm is the computation of the initial disparity image.

A.5 MD

Molecular Dynamics (MD) context is in general a computer simulation of atoms or molecules which derives from N-Body simulation. In this particular application we simulate the behaviour of liquid Argon molecules restricted in a bounded box. Specifically we simulate kinematic properties such as position, velocity etc of liquid Argon atoms when they act in a kind of force produced by a Lennard-Jones pair potential [14]. MD simulations find appliance in many science domains such as theoretical physics, biochemistry and biophysics. For example they are used to examine atomic-level effects of dynamics, that cannot be observed with naked eye or any other macroscopic technique, such as ion-subplantation.

A.6 BONDS

Bonds [13] is library containing applications of bond-related computations. Bond is a loan that exists between an issuer and a holder. The issuer is obligated to pay the holder the initial loan augmented by an interest.

B

MOLECULAR DYNAMICS

Listing B.1: Molecular Dynamics written in Centaurus programming model

```
for(int i = 0; i < particlesnumber; i++) {

        float x = ((float)_rand())/((float)(32767))*max_pos -
            min_pos;
        float z = ((float)_rand())/((float)(32767))*max_pos -
            min_pos;
        float y = ((float)_rand())/((float)(32767))*max_pos -
            min_pos;
        pos[i] = f4(x, y, z, 1.f);

        x = ((float)_rand())/((float)(32767))*2.0f - 1.0f;
        z = ((float)_rand())/((float)(32767))*2.0f - 1.0f;
        y = ((float)_rand())/((float)(32767))*2.0f - 1.0f;
        vel[i] = f4(x, y, z, 0.f);

        force[i] = f4(0.f, 0.f, 0.f, 0.f);

        color[i] = f4(1.0f, 0.0f, 0.0f, 1.0f);
}

max_steps = (int)duration/dt;

for (int run=0; run<=max_steps; run++) {

        mod2 = run%2;
        pos_in = ( mod2 == 0)? pos: pos_buf;
        pos_out = ( mod2 == 0)? pos_buf: pos;

    force_in = ( mod2 == 0)? force: force_buf;
        force_out = ( mod2 == 0)? force_buf: force;

        vel_in = ( mod2 == 0)? vel: vel_buf;
        vel_out = ( mod2 == 0)? vel_buf: vel;

        potential_in = ( mod2 == 0)? potential: potential_buf;
        potential_out = ( mod2 == 0)? potential_buf: potential;

        virial_in = ( mod2 == 0)? virial: virial_buf;
```

```
virial_out = ( mod2 == 0)? virial_buf: virial;

#pragma acl task label("0") approxfun(update_Approx)
    device_in(pos_in, force_in, vel_in, potential_in,
    color, virial_in)
    device_out(pos_out[0:particlesperdevice],
    vel_out[0:particlesperdevice],
    potential_out[0:particlesperdevice],
    force_out[0:particlesperdevice],
    virial_out[0:particlesperdevice]) significant(100)
    workers(localwork[0],localwork[1])
    groups(globalwork[0],globalwork[1])
update(pos_in, color, force_in, vel_in, potential_in,
    bound, dt, particlesnumber, virial_in, offsets[0],
    pos_out, vel_out, force_out, potential_out,
    virial_out);

#pragma acl task label("1") approxfun(update_Approx)
    device_in(pos_in, force_in, vel_in, potential_in,
    color, virial_in)
    device_out(pos_out[offset[1]:particlesperdevice],
    vel_out[offset[1]:particlesperdevice],
    potential_out[offset[1]:particlesperdevice],
    force_out[offset[1]:particlesperdevice],
    virial_out[offset[1]:particlesperdevice])
    significant(100) workers(localwork[0],localwork[1])
    groups(globalwork[0],globalwork[1])
update(pos_in, color, force_in, vel_in, potential_in,
    bound, dt, particlesnumber, virial_in, offsets[1],
    pos_out, vel_out, force_out, potential_out,
    virial_out);

#pragma acl task label("2") approxfun(update_Approx)
    device_in(pos_in, force_in, vel_in, potential_in,
    color, virial_in)
    device_out(pos_out[offset[2]:particlesperdevice],
    vel_out[offset[2]:particlesperdevice],
    potential_out[offset[2]:particlesperdevice],
    force_out[offset[2]:particlesperdevice],
    virial_out[offset[2]:particlesperdevice])
    significant(100) workers(localwork[0],localwork[1])
    groups(globalwork[0],globalwork[1])
update(pos_in, color, force_in, vel_in, potential_in,
    bound, dt, particlesnumber, virial_in, offsets[2],
    pos_out, vel_out, force_out, potential_out,
    virial_out);

#pragma acl task label("3") approxfun(update_Approx)
    device_in(pos_in, force_in, vel_in, potential_in,
    color, virial_in)
    device_out(pos_out[offset[3]:particlesperdevice],
    vel_out[offset[3]:particlesperdevice],
```

```
            potential_out[offset[3]:particlesperdevice],
            force_out[offset[3]:particlesperdevice],
            virial_out[offset[3]:particlesperdevice])
            significant(100) workers(localwork[0],localwork[1])
            groups(globalwork[0],globalwork[1])
        update(pos_in, color, force_in, vel_in, potential_in,
            bound, dt, particlesnumber, virial_in, offsets[3],
            pos_out, vel_out, force_out, potential_out,
            virial_out);

        #pragma acl taskwait


}
```

BIBLIOGRAPHY

[1] G. E. Moore. Cramming more components onto integrated circuits. Electronics, 38(8), April 1965.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. IEEE Journal of Solid-State Circuits, 9, October 1974.

[3] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pages 198-209, New York, NY, USA, 2010. ACM.

[4] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. A variability-aware openmp environment for eficient execution of accuracy-configurable computation on shared-fpu processor clusters. In Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '13, pages 35:1-35:10,Piscataway, NJ, USA, 2013. IEEE Press.

[5] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 164-174, New York, NY, USA, 2011. ACM.

[6] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test, 12(3):6673, May 2010.

[7] OpenACC standard committee. The OpenACC Application Programming Interface, v2.0, June 2013.

[8] NVIDIA. NVML API Reference. http://docs.nvidia.com/deploy/nvml-api/index.html.

[9] Intel. Intel 64 and ia-32 architectures software developer manual, 2010. Chapter 14.9.1.

[10] V. Prisacariu and I. Reid. fastHOG-a real-time GPU implementation of HOG. Technical Report 2310/9, Department of Engineering Science, Cambridge University, 2009.

[11] X. Feng, K. W. Cameron, and D. A. Buell. Pbpi: A high performance implementation of bayesian phylogenetic inference. In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, New York, NY, USA, 2006. ACM.

[12] K. Yamaguchi, D. McAllester, and R. Urtasun. Efficient joint segmentation, occlusion labeling, stereo and flow estimation. In ECCV, 2014.

[13] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos. Accelerating financial applications on the gpu. In Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, pages 127–136. ACM, 2013.

[14] Jones, J. E. On the Determination of Molecular Fields. I. From the Variation of the Viscosity of a Gas with Temperature. Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, pages 441-462. The Royal Society, 1924

[15] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[16] M. Ringenburg, A. Sampson, I. Ackerman, and L. C. D. Grossman. Monitoring and debugging the quality of results in approximate programs. In Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), Istanbul, Turkey, March 2015.

[17] Q. Zhang, F. Yuan, R. Ye, and Q. Xu. Approxit: An approximate computing framework for iterative methods. In Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference, DAC '14, pages 97:1–97:6, New York, NY, USA, 2014. ACM.

[18] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. A variability-aware openmp environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters. In Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '13, pages 35:1–35:10, Piscataway, NJ, USA, 2013. IEEE Press.

[19] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. ACM Trans. Embed. Comput. Syst., 12(2s):88:1–88:26, May 2013.

[20] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. Sage: Self-tuning approximation for graphics engines. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[21] X. Li. Power Management for GPU-CPU Heterogeneous Systems. Master's thesis, Uni- versity of Tennessee, 12 2011.

[22] K. H. Tsoi and W. Luk. Power profiling and optimization for heterogeneous multi-core systems. SIGARCH Comput. Archit. News, 39(4):8–13, Dec. 2011.